VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra Informatiky

Simulátor 3D terénu
3D terrain

2010 Jaromír Krpec

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě dne Podpis

…..…… …..……

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Martinu Němcovi, Ph.D. za podnětné připomínky a cenné rady při vytváření praktické i teoretické části této bakalářské práce.

Abstrakt

Práce je zaměřena na procedurální generování povrchu terénu. Základem je nastudování základních algoritmů pro generování, úpravu a zobrazení terénu. Stěžejní část práce se zabývá možnostmi generování náhodného terénu, součástí je popis vybraných algoritmů pro optimalizaci a urychlení vykreslování scény. Všechny zvolené algoritmy byly prakticky naimplementovány tak, aby si je mohl uživatel otestovat. Nakonec jsou zde také prezentovány postupy, které ukazují jak optimalizovat výslednou scénu a urychlit vykreslování. Součástí práce je aplikace, ve které jsou realizovány popsané algoritmy. Program je možné používat jako applet v internetovém prohlížeči.

Kličová slova

terén, OpenGL, JOGL, Quad-tree, Frustum culling, OBJ, skydome, procedurální generování textur, lighmapping, multitexturing

Abstract

This work is dealed with procedural generation of terrain surface. The key is study of basic algorithms for generation, edit and rendering terrain. Main part of this work is deal with options of generation random terrain, also description selected optimalization algorithms for speed up rendering of scene. All from these algorithms was implemented for testing by user. At the end I have presented processes, which show how is possible to optimalize result scene and speed up rendering. The part of this work is also an application where I implemented described algorithms. The application can be used as an applet in the web browser.

Keywords

terrain, OpenGL, JOGL, Quad-tree, Frustum culling, OBJ, skydome, procedural texture generation, lighmapping, multitexturing

Obsah

1. Úvod

V oboru počítačové grafiky má generování terénu již dlouhou historii, během které postupně proniklo do mnoha odvětví. Uživatelé se mohou setkat s počítačově vygenerovanými krajinami například ve vojenství, letectví, stavebnictví apod. Příkladem použití mohou být například letecké simulátory používané pro výuku a trénink pilotů.

Vedle generování povrchu terénu se tato práce zabývá také jeho optimalizacemi a vykreslováním. Obecně je vykreslování možné provádět softwarově nebo využít hardwarově akcelerované možnosti grafických karet využitím příslušných knihoven například OpenGL nebo DirectX.

Přestože jsou zpřístupněny možnosti hardwarově akcelerované grafiky, které mohou vést ke zrychlení aplikace, je vykreslování stále velmi náročné na výkon. Proto je vhodné se zabývat algoritmy, které optimalizují generování, optimalizaci i zobrazení scény.

První část se zabývá popisem algoritmů pro procedurální generování terénu. Dále se práce zabývá metodami pro vykreslování 3D grafiky, procedurální generování textur na základě výšky a používání lightmap. Součástí práce je i možnost načíst a zobrazit ve scéně 3D modely ve formátu OBJ. Pro reálnější zobrazování bylo nutné doplnit práci o algoritmy pro osvětlení scény.

2. Generování terénu

K reprezentaci povrchů krajin v počítačové grafice se zpravidla používá datová struktura jednorozměrného nebo dvourozměrného pole čísel, které se označuje jako výšková mapa. Velikost tohoto pole je pak dána součinem rozměrů krajiny a jednotlivé prvky v něm představují výšku v určitém bodě. [1]

Výškové mapy se mohou samozřejmě znázornit i v rovině například v odstínech šedi, kde světlejší místa mohou být území s nižší výškou a tmavší místa jsou naopak vyšší oblasti. Načte-li se výšková mapa do paměti, pak je v paměti uložena pouze souřadnice představující právě výšku, neboť zbylé dvě souřadnice pro každý bod jsou vytvářeny při vykreslování jako indexy výškové mapy, které jsou potřeba znát pro přístup k požadovanému prvku pole.

Výhodou výškových map je zejména snadný přístup k prvkům při generování, vykreslování a mapování textur. Na druhou stranu jejím velkým nedostatkem je, že nelze pomoci ní reprezentovat výskyt některých přírodních útvarů, jako jsou převisy či jeskyně. Je tomu tak proto, že nelze pomoci výškových map znázornit pro jeden konkrétní bod v rovině XY více než jednu výšku. Z tohoto důvodu se někdy označuje jako 2.5D.

Chceme-li vytvořit reálně vypadající krajinu, obvykle nelze použít obyčejný generátor náhodných čísel. Při vytváření terénu tímto způsobem by došlo k nežádoucím vlastnostem, jakými by byly především nepřirozené zlomy, neboť mezi sousedními prvky není žádný vztah, který by těmto jevům mohl zabránit. Nicméně v praxi je možno se i s tímto přístupem setkat, především tehdy, je-li kladen požadavek na vytváření kráterovitých povrchů.

Velmi často se pro generování terénů, ale i jiných přírodních jevů a objektů, využívají fraktály [10] umožňující vygenerovat terén pomoci jednoduchého algoritmu, který vede k vytvoření, zejména ve srovnání s pouhým generováním náhodných čísel, mnohem reálnějšího povrchu terénu.

V závislosti na situaci, kdy je požadován například kopcovitý či schodovitý typ terénu, je možné zvolit algoritmus, pomoci kterého můžeme požadovaný typ vygenerovat, díky odlišným výstupům jednotlivých algoritmů.

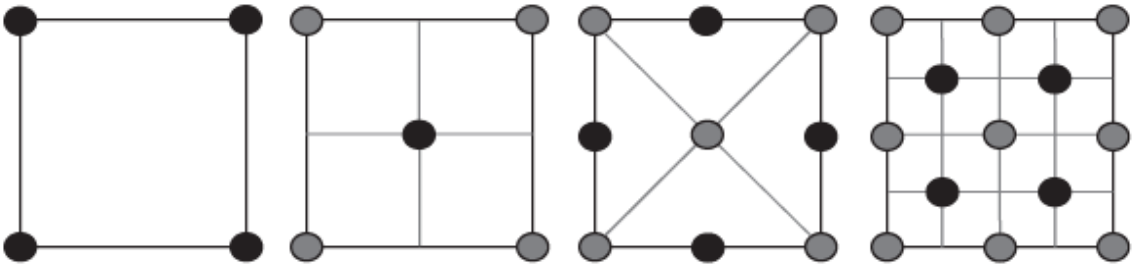
2.1. Algoritmus přesouvání středového bodu (midpoint displacement)

Principem algoritmu je nalezení bodu, který se nachází uprostřed určité části terénu a přičtení náhodné hodnoty.

Průběh algoritmu lze jednoduše znázornit v situaci, kdy je možné aplikovat algoritmus přesouvání středového bodu na úsečku, jak je ukázáno na obrázku 2.1. Je-li dána úsečka AB, pak je jako první nalezen střed úsečky, který označíme jako bod C a přičteme pro něj náhodnou hodnotu. Na úsečce tak vzniknou dva stejně velké segmenty AC a CB, pro které jsou v další iteraci nalezeny středy D a E, čímž vzniknou čtyři segmenty AD, DC, CE, EB.

Obr. 2.1.:grafické znázornění průběhu algoritmu přesouvání středového bodu na úsečce

Pro generování výškové mapy se algoritmus aplikuje na čtverec. První dvě iterace jsou znázorněny na obrázku 2.2.



Obr. 2.2.:grafické znázornění průběhu algoritmu přesouvání středového bodu

Podmínkou je, že výšková mapa bude čtvercová, nicméně tvar vykresleného terénu čtvercový být nemusí. V rozích výškové mapy bude implicitně nastavena určitá hodnota. Jsou-li body $P_1[x_0, f(x_0, y_0), y_0]$, $P_2[x_n, f(x_n, y_0), y_0]$, $P3[x_0, f(x_0, y_n), y_n]$, $P4[x_n, f(x_n, y_n), y_n]$ rohové, pak právě jejich funkční hodnota je rovna defaultním hodnotám.

Výpočet dále pokračuje tak, že se najde prostřední bod čtverce $P_s[x_{1/2}, f(x_{1/2}, y_{1/2}), y_{1/2}]$ tím, že se určí středy dvou stran:

$$x_{1/2} = \frac{x_0 + x_n}{2}$$

$$y_{1/2} = \frac{y_0 + y_n}{2}$$

Spočítá se průměrná výška ze čtyř již známých rohových bodů a tento bod se ještě zvětší o číslo δ, které je gaussovským náhodným číslem. Velikost tohoto náhodného čísla se mění na základě hloubky rekurze.

$$f(x_{1/2}, y_{1/2}) = \frac{f(x_0, y_0) + f(x_n, y_0) + f(x_0, y_n) + f(x_n, y_n)}{4} + \delta$$

V dalším kroku se vypočítají body, jež se nacházejí na hranách mapy a to tak, že je otočena o 45° a stejným způsobem se vypočítají hodnoty bodů ve středech hran. Někdy se tento algoritmus označuje diamond-square algorithm, neboť toto otáčení o 45° vizuálně připomíná diamant.

Nakonec je pak otočena zpět o 45°. Z jedné větší části se tak vytvořili čtyři menší. Výpočet hodnot středového bodu a bodů, jež se nacházejí na hranách, je provedeno v každém nově vzniklém čtverci výškové mapy. První dvě iterace jsou znázorněny na obrázcích 2.1 a 2.2.

Obrázek 2.3 ukazuje výsledné výškové mapy převedené do odstínu šedi, kde tmavší místa jsou oblasti s vyšší výškou, v závislosti na hodnotě vstupní konstanty *k*, jenž může nabývat hodnot v rozmezí 0.0 až 1.0.

Obr. 2.3.: vliv vstupní konstanty na výslednou výškovou mapu pro hodnoty k = 0,1(vlevo), k=0,5(uprostřed) a k=1,0(vpravo)

Vstupem algoritmu je pouze velikost výškové mapy a konstanta *k* určující drsnost terénu.

Velikost povrchu krajiny musí být $2^n + 1$. Pak paměťová složitost je $O(2^n + 1)^2$ a časová složitost algoritmu je $\log N$, kde N je hloubka rekurze.

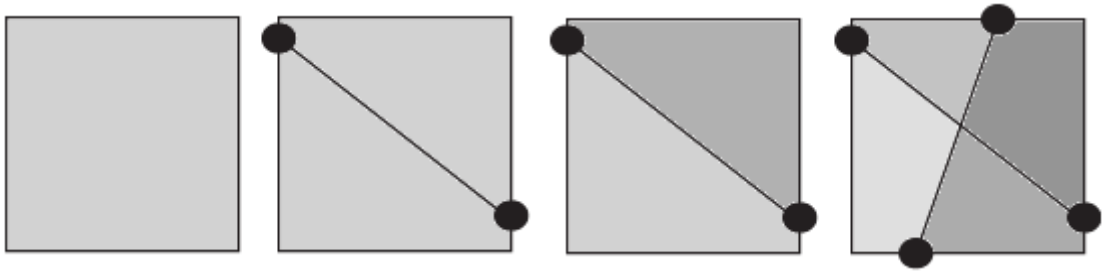
2.2. Algoritmus náhodných poruch (random faults)

Algoritmus pracuje na velmi jednoduchém principu, kdy je vytvořen zlom, jenž je možno si velmi dobře představit jako výsledek procesu, při kterém jsou vždy, v rámci jedné iterace, vybrány dva náhodné body v krajině, které jsou následně spojené přímkou, čímž dojde k rozdělení krajiny na dva celky.

Opět lze algoritmus znázornit pomoci úsečky. Průběh je na obrázku 2.4, kdy je na úsečce AB nalezen bod, který rozděluje úsečku na dvě části a následně je jedné části přičtena náhodná hodnota a druhé části odečtena.

Obr. 2.4.: grafické znázornění průběhu algoritmu náhodných poruch na úsečce

Průběh pro generování výškové mapy:



Obr. 2.5.: grafické znázornění průběhu algoritmu náhodných poruch

Algoritmus začíná nejprve nastavením hodnoty 0 všem bodům na výškové mapě. Dále dojde k určení zlomu a rozdělení krajiny na dvě části. V žádném případě není nutné, aby byly vzniklé části stejně velké. Všem bodům v jednom z těchto dílů je přičtena určitá náhodná hodnota. V druhém dílu je pak naopak výška o tuto hodnotu odečtena. Po skončení každé iterace, se náhodné číslo postupně zmenšuje. Na obrázku 2.5 je možné pozorovat první dvě iterace algoritmu.

Při použití tohoto způsobu generování terénu je vhodné nastavit přiměřený počet opakování, aby byl výsledek co možná nejlepší. Na počtu iterací pak závisí nejen výsledný terén, ale i délka výpočtu.

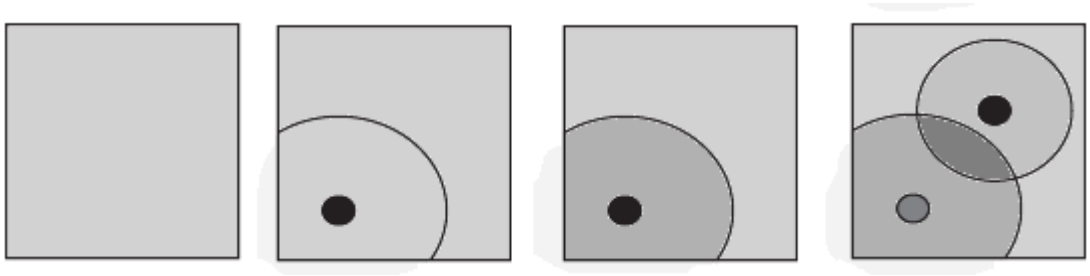
Obr. 2.6.: vliv počtu iterací na výslednou výškovou mapu. Počet iterací i = 5(vlevo), i=50(uprostřed) a i=250 (vpravo)

Vstupem algoritmu je velikost výškové mapy, minimální velikost zlomu, maximální velikost zlomu a počet iterací.

Velikost mapy je $m \cdot n$. Paměťová složitost je $O(m \cdot n)$ a časová složitost je $I \cdot m \cdot n$, kde I je počet iterací.

2.3. Algoritmus generování kopců (circles algorithm)

Hlavní myšlenkou tohoto algoritmu je, že na náhodná místa ve výškové mapě jsou umisťovány rotační paraboloidy s náhodnou výškou a poloměrem.

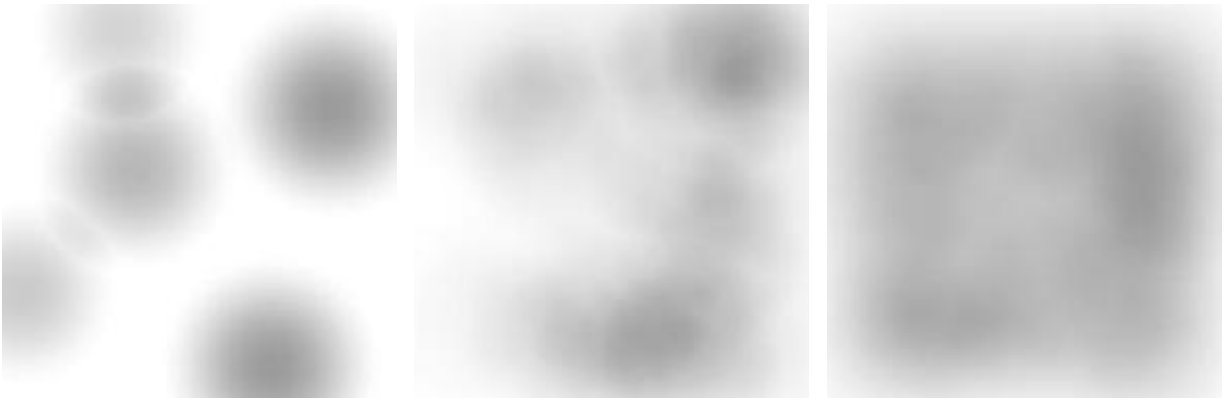


Obr. 2.7.: grafické znázornění průběhu algoritmu generování kopců

V každé iteraci se nejprve najde střed, jenž se může nacházet kdekoliv v terénu a také se určí poloměr, u kterého je možné uživateli umožnit zadávání omezující podmínky, což platí i pro maximální výšku.

Jakmile jsou známy tyto dva podstatné příznaky, je možné pro všechny body, nacházející se uvnitř vytvořeného kruhu, přičíst hodnotu, kterou lze získat ze vzorce pro rotační paraboloid.

Výsledný terén nemusí působit vždy tak realisticky jako při použití algoritmů využívající fraktály, přesto je možno i použitím tohoto algoritmu, především při vhodně zvoleném počtu opakování, získat dobré výsledky.



Obr. 2.8.: vliv počtu iterací na výslednou výškovou mapu. Počet iterací i = 5(vlevo), i=50(uprostřed) a i=250 (vpravo)

Vstupem algoritmu je velikost výškové mapy, počet iterací, minimální a maximální možná velikost poloměru a maximální výška rotačního paraboloidu.

Velikost mapy je $m \cdot n$. Paměťová složitost je $O(m \cdot n)$

2.4. Vyhlazovací filtr

Ve většině případů, i při použití fraktálních algoritmů, se ve výsledném terénu objevují nepřirozené lomy a špičaté vyvýšeniny, tudíž je možné po vytvoření terénu na jeho povrch aplikovat vyhlazování.

Lze využít metodu závislou na jednom sousedním bodě, kdy dochází k interpolaci vybraného bodu pomoci parametru q, jenž určuje velikost vlivu hodnoty sousedního bodu na aktuálně procházený bod:

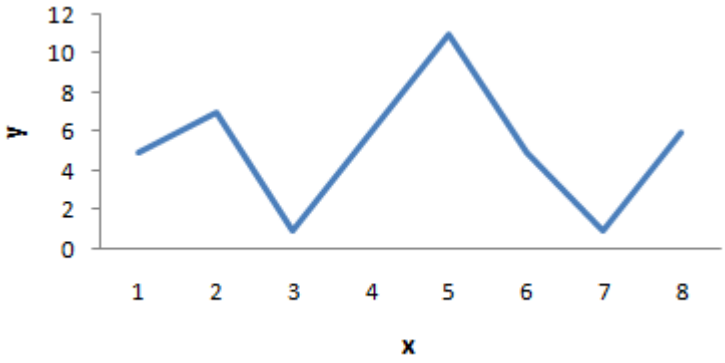
$$heightmap[x][y] = heightmap[x-1][y] * (1-q) + heightmap[x][y] * q$$

Parametr q nabývá hodnot 0 až 1. Terén zůstane beze změny, pokud q je rovno 1. Naopak odpovídá-li q hodnota 0, pak hodnota aktuálně vybraného bodu je nahrazena jeho sousedním vrcholem.

Vyhlazování je možné aplikovat několikrát po sobě. Je také možné aplikovat jej na řádky i sloupce výškové mapy.

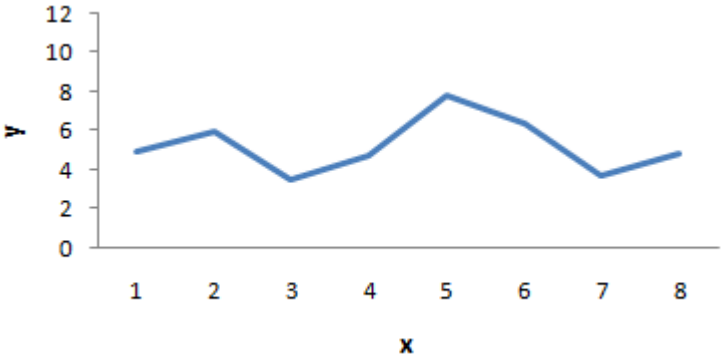
Jelikož je pracováno již s vytvořenou výškovou mapou, která se jen upravuje, tak paměťová složitost je závislá na rozlišení výškové mapy. Časová složitost je navíc závislá na počtu opakování filtrace.

1	5
2	7
3	1
4	6
5	11
6	5
7	1
8	6



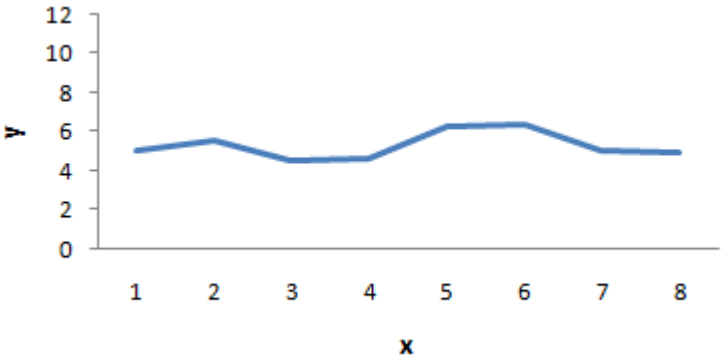
Graf 2.1.: Povrch terénu bez aplikování filtru

1	5
2	6
3	3,5
4	4,75
5	7,875
6	6,4375
7	3,7188
8	4,8594



Graf 2.2.: Povrch terénu po první iteraci filtru

1	5
2	5,5
3	4,5
4	4,625
5	6,25
6	6,3438
7	5,0313
8	4,9454



Graf 2.3.: Povrch terénu po druhé iteraci filtru

Lze však aplikovat i jiné filtry. Na rozdíl od předchozího, jenž k výpočtu používá jen jeden sousední bod, pak maticový filtr může použít více sousedních bodů. V případě matice o rozměrech 3x3 je k vyhlazení použito 8 sousedních výškových bodů a ve středu se nachází aktuálně zpracovávaný bod. [5]

3. Vykreslování terénu

3.1. Úvod

OpenGL (Open Graphics Library) [9] je standardem navrženým firmou SGI roku 1992 popisující neobjektové multiplatformní rozhraní pro tvorbu 2D nebo 3D počítačové grafiky, jehož základní funkcí je použití frame bufferu.

Rozhraní bylo navrženo tak, aby bylo možné knihovnu použít v jakémkoliv programovacím jazyce, díky čemuž existuje celá řada rozšíření pro stávající jazyky. Původně byl však určen pro použití v jazycích C a C++.

Funkce, které OpenGL poskytuje, umožňují programátorovi vykreslovat tělesa složená z různých primitivních geometrických prvků (úsečka, trojúhelník, polygon apod.). Součástí knihovny jsou hardwarově podporované algoritmy, což umožňuje usnadnění a urychlení výsledného vykreslování (např. při výpočtu osvětlení, texturování, řešení viditelnosti a v mnoha dalších směrech).

Zároveň je možné použít knihovnu GLU (OpenGL Utility library) představující nadstavbu OpenGL, ve které je implementována celá řada pokročilejších algoritmů. Díky GLU lze velmi jednoduše vytvářet základní kvadriky (koule, elipsoidy,…), pracovat s NURBS křivkami a plochami a mnoho dalšího.

Vedle OpenGL existuje ještě často používaná knihovna Direct3D, která je součástí balíku DirectX [8] vyvíjená společností Microsoft k zajištění vysokého výkonu grafických aplikací, především počítačových her, pro operační systém Windows. Funkce, které poskytuje, jsou však v zásadě totožné s těmi, které nabízí OpenGL.

V podstatě má dnes programátor na vybranou právě z těchto dvou rozhraní, kdy u OpenGL je třeba vzít na vědomí odlišné přizpůsobení pro různé jazyky.

Pro implementaci programu k této práci byla však vybrána knihovna OpenGL. Ne jen proto, že je standardem, ale především pro multiplatformní vlastnosti a možnosti nasadit aplikaci na internet jako Java Applet.

3.1.1. JOGL

JOGL [7] je jedna z několika knihoven, která přináší možnost využívat v 3D grafických aplikacích napsaných v jazyce Java možnosti rozhraní OpenGL.

Nespornou výhodou oproti jiným knihovnám, které umožňují používat 3D grafiku v Javě, je možnost používání funkcí OpenGL stejně jako v jazyku C/C++, neboť autoři z důvodu dodržet objektový přístup pouze umístili jednotlivé funkce do jedné třídy s názvem GL, popřípadě funkce pro rozšíření OpenGL Utility Library je možné najít ve třídě GLU.

Implementovaná aplikace využívá verzi JOGL 1.1.1a, ale již existuje verze JOGL 2.0 beta, která by měla přinášet kompatibilitu s OpenGL 3.0.

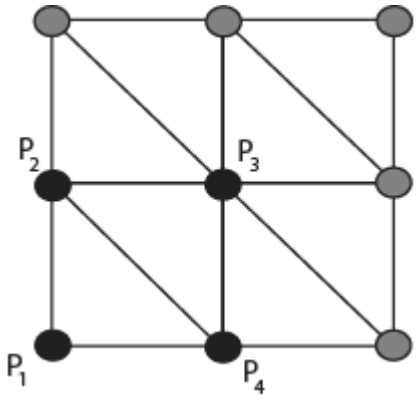
3.2. Vykreslení pomoci trojúhelníkové sítě

V okamžiku, kdy je terén nejprve vygenerován některým z algoritmů, je možné jej zobrazit pomoci trojúhelníkové sítě, do které je výšková mapa převedena.

Pravidelnou síť lze vytvořit pomoci dvou do sebe vnořených cyklů procházejících výškovou mapu. Výškový bod je pak vybrán dle aktuálních hodnot proměnných probíhajícího cyklu, které představují indexy pole a zároveň zbylé dvě souřadnice.

Takto jsou získány všechny tři složky bodu prostoru. Nicméně je však nutné získat souřadnice dalších tří bodů, aby bylo možno vykreslit celý pás trojúhelníků, protože princip vytváření pravidelné

trojúhelníkové sítě spočívá ve vytvoření dvou trojúhelníků mezi každými čtyřmi výškovými body, jak ukazuje obrázek 3.1.



Obr. 3.1.: část pravidelné trojúhelníkové sítě

Souřadnice těchto čtyř bodů jsou:

- P1 [x, heightMap[x][y], y]
- P2 [x+1, heightMap[x+1][y], y]
- P3 [x, heightMap[x][y+1], y+1]
- P4 [x+1, heightMap[x+1][y+1], y+1]

Krom pravidelné sítě lze pro reprezentaci terénu použít i nepravidelnou trojúhelníkovou síť. Její výhoda spočívá v možnosti pro více členitá místa použít větší množství trojúhelníků než v místech s malými změnami povrchu.

3.3 Texturování

Texturováním je označována technika, která umožňuje nanášet dvourozměrný obrázek na povrch určitého objektu. Důvodem texturování je dodání realistického vzhledu, čímž se celková podoba celé scény stane vizuálně mnohem skutečnější.

3.3.1. Procedurální generování textury povrchu

Existuje samozřejmě více metod, jak mapovat povrch terénu. Pravděpodobně nejjednodušší způsob je celý povrch pokrýt jednou texturou, která může například znázorňovat trávu. V reálném světě však bývá krajina velmi rozmanitá, ve které se může vyskytovat nejen tráva, ale i písek, skály, sníh a mnoho dalších nejrůznějších elementů.

Další možností je použít předem určeného množství textur, kdy každá z nich reprezentuje jiný povrch. Vlastní texturování pak probíhá tak, že se každý trojúhelník terénu na základě výšky pokryje odpovídající texturou. Problém této metody je, že při přechodu mezi dvěma texturami vznikají ostré a tudíž nepřirozené přechody kopírující tvar trojúhelníku, na který byl obrazec mapován.

Procedurální generování textury povrchu je postup mapování povrchu, který se dokáže velmi dobře přizpůsobit terénu, jenž byl dříve náhodně vygenerován.

Podobně jako předchozí metoda je třeba mít k dispozici několik textur, jež znázorňují různé úrovně v krajině. Pro každou z nich se určí, v jakém výškovém rozmezí se bude která textura vyskytovat, čímž dojde k rozdělení na určitý počet regionů.

Důležitou součástí algoritmu je funkce, která určí procentuální zastoupení jednotlivých textur v každém místě terénu v závislosti na výšce. Je-li například určeno, že tráva se vyskytuje ve výšce 64-92, pak ve výšce 64 je textura trávy zastoupena na 0%, zatímco ve výšce 92 je na 100%. Čím blíže se

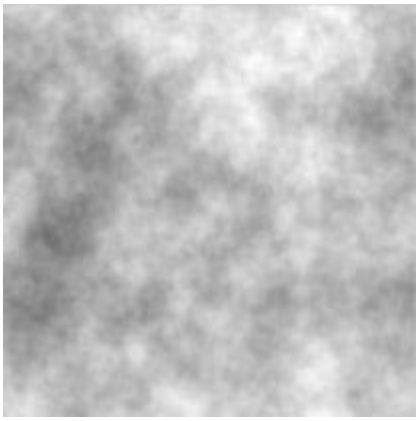
probíhající vytváření textury blíží k místu dělení, tím více se textury jedna s druhou začínají promíchávat.

Takto jsou poměrně jednoduše vyřešeny naprosto přirozené pozvolné přechody mezi jednotlivými předdefinovanými texturami, protože dochází k mixování barevných bodů obou zúčastněných textur a vytvoření jedné výsledné textury, kterou se pokryje celý povrch terénu, popřípadě větší část terénu. Technikou procedurálního generování textur se dá dospět k mnohem lepším výsledkům, než při použití některých sice jednodušších i rychlejších, ale ve výsledku velmi nedostačujících metod.

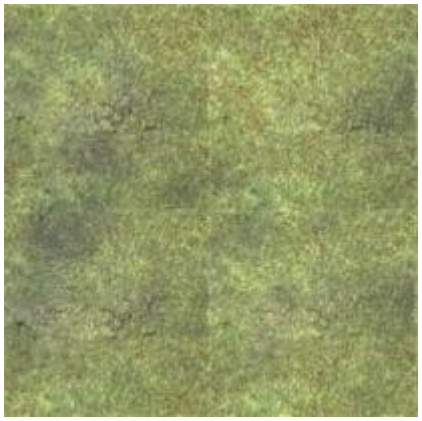
Jsou-li například použity pro jednotlivé stupně textury z obrázku 3.2.

Obr. 3.2.: sada zdrojových textury

Pak použitím právě popisovaného algoritmu procedurálního generování textur vznikne textura na obrázku 3.4 v závislosti na vstupní výškové mapě, která se nachází na obrázku 3.3.



Obr. 3.3.: vstupní výšková mapa



Obr. 3.4.: výsledná textura povrchu

3.3.2. Multitexturing

Grafické karty v dnešní době již standardně podporují techniku multitexturing, což umožňuje z pohledu programátora při použití knihovny OpenGL velmi jednoduše mapovat dvě či více textur na jeden objekt ve stejný okamžik při jednom průchodu, čímž se docílí zvýšení výsledné vizuální kvality. Názornost rozdílu je na obrázcích 3.5 a 3.6.

Požitím více textur na povrch vytvořeného terénu může způsobovat například dojem zdrsnění povrchu, čímž se velmi zvýší realističnost celého objektu, neboť pokud se na terén mapuje jen procedurálně vygenerovaná textura, tak působí hladce a budí dojem ne příliš reálného vzezření spíše připomínající plastový model. Použití multitexturingu je zároveň možno využívat i při aplikování lightmap.

Obr. 3.5.: bez použití multitexturingu

Obr. 3.6.: s použitím multitexturingu

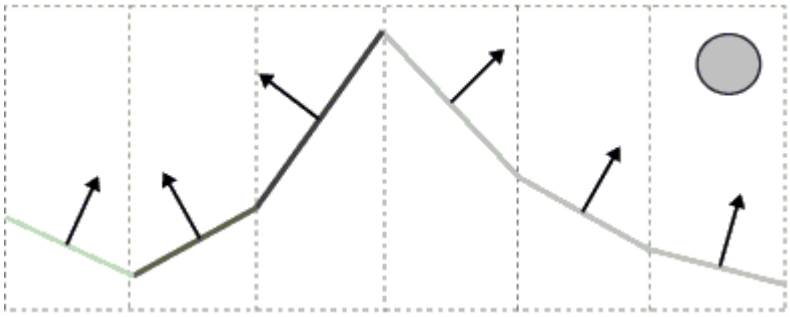
3.3.3. LightMapping

Lightmapy jsou textury, které obsahují v barevných bodech informaci o jasu určitého povrchu. Právě proto bývají jednotlivé pixely pouze v odstínech šedi. Velmi jednoduše se tak dají znázornit místa tělesa, která jsou osvětlená a která nikoliv.

Použití může nalézt také pro dynamické osvětlení, kdy se v závislosti na měnící se pozici světelného zdroje, vždy vygeneruje nová lightmapa. Například pro pohybující se Slunce osvětlující krajinu by se buď předem, popřípadě za běhu vygenerovalo několik takovýchto textur, kdy by postupně jedna druhou nahrazovala. Především se ale používá na plochy, na které dopadá světlo ze zdroje, jehož pozice je statická.

Výhodou jejich použití je především rychlost, neboť odpadá nutnost počítat při vykreslování u vytvořeného terénu normály, případně plnit jimi paměť počítače.

Postup, jenž byl použit při implementaci, je poměrně jednoduchý a je znázorněn na obrázku 3.7, který ukazuje závislost normálových vektorů na pozici zdroje a určení tak zastíněné a osvětlené plošky.

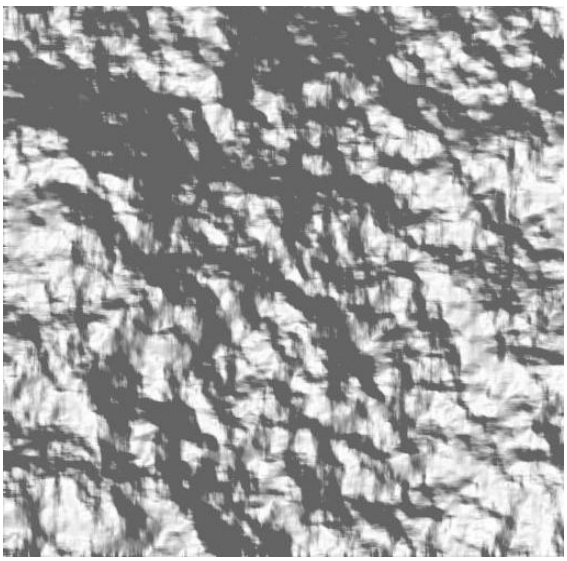


Obr. 3.7.: Vytvoření lightmapy podle normál

Postupně je procházen celý terén a pro každý trojúhelník se spočítá normálový vektor. Zároveň se také určí vektor paprsku světelného zdroje na základě pozice trojúhelníku a zdroje světla. A nakonec v závislosti na úhlu, který svírá paprsek s normálou, se určí zastínění aktuálního pixelu. Pokud je trojúhelník odvrácený, pak na něj světlo nedopadá a bude tmavší než ten, na jehož povrch světlo dopadne.

Tato metoda však nepočítá vržené stíny, které mohou vzniknout například v okamžiku, kdy kopec zastiňuje údolí.

Na obrázku 3.8 je ukázka vytvořené lightmapy.



Obr. 3.8.: Lightmapa

3.4. SkyDome

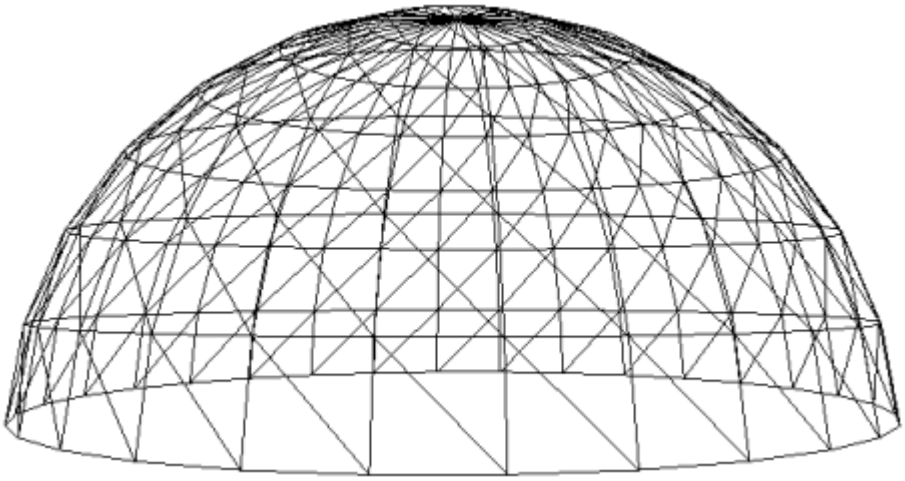
Pro zvýšení celkového vizuálního dojmu scény je vhodné doplnit scénu o oblohu. Jednou z možností je obklopit model terénu určitým objektem, jenž představuje oblohu.

Objekt reprezentující oblohu, může být krychle (skybox). V tomto případě se na každou plochu krychle mapuje textura. Je tedy zapotřebí šest textur, které na sebe musí navazovat, neboť je důležité, aby nebyli viditelné hrany.

Výhodou použití skyboxu jsou nízké požadavky na počet trojúhelníků, protože na každou plochu je zapotřebí pouze dvou trojúhelníků. Celkem je nutné vykreslit jen 12 trojúhelníků. S malým množstvím trojúhelníků souvisí i jednoduché určení texturových koordinátu pro mapování textur.

Na druhou stranu je nevýhodou nutnost používání všech šesti textur s vysokým rozlišením, aby bylo vytvořené pozadí co možná nejdetailnější. Navíc ne vždy na sebe textury korektně navazují.

SkyDome je komplikovanější, avšak mnohem reálnější způsob, jak reprezentovat oblohu. Hlavním principem je vytvoření polokoule, na kterou se posléze mapuje textura oblohy. Díky tvaru a především nanesením textury nejsou i při ne příliš velkém množství trojúhelníků, ze kterého se skládá, vidět ostré hrany či lomy.



Obr. 3.9.: trojúhelníková síť skydomu

Výpočet jednotlivých složek bodů všech trojúhelníků, ze kterých je polokoule vytvořena, vychází z parametrického vyjádření koule:

$$x = x_0 + r \sin\theta \cos\varphi$$

$$y = y_0 + r \sin\theta \sin\varphi$$

$$z = z_0 + r \cos\theta$$

Pro algoritmus je také důležité vědět, kolik je potřeba alokovat místa v paměti pro uložení bodů trojúhelníků. Výpočet je velmi jednoduchý a výsledek závisí především na proměnné h, která představuje požadované zaoblení výsledné polokoule. Čím je h menší, tím je potřeba alokovat více paměti, zato však je objekt kulatější.

V případě skydomu není nutné jej příliš zakulacovat, neboť ostré hrany při pohledu uvnitř polokoule díky namapování textury zanikají. Je třeba pamatovat, že velké množství trojúhelníků, ze kterého by se skládal, by vedlo k větší náročnosti aplikace.

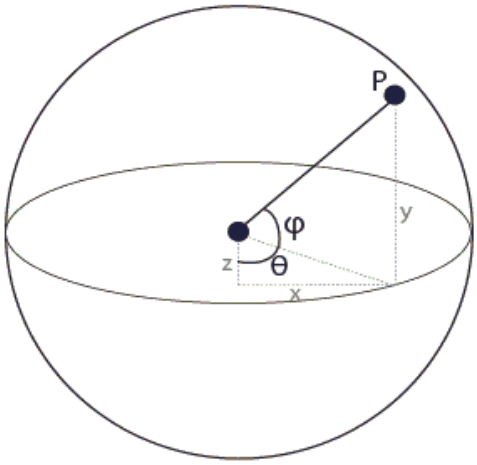
$$Počet\ bodů = \left(\frac{360}{h}\right) * \left(\frac{90}{h}\right) * 4$$

Pseudokód určení jednotlivých bodů:

```
for φ = 0 to PI/2, φ += Δφ
{
    for θ = 0 to 2*PI, θ += Δθ
    {
        bod_x = poloměr * sin(φ) * cos(θ)
        bod_y = poloměr * sin(φ) * sin(θ)
        bod_z = poloměr * cos(φ)
    }
}
```

Následně se vypočítají koordináty důležité pro mapování textury na každý bod trojúhelníku, které lze odvodit pomoci goniometrických funkcí, přičemž je nutné zajistit, aby výsledná hodnota byla v rozmezí 0-1.

V prvním kroku se pro každý bod určí vektor mezi tímto bodem a počátkem. Poté se vektor normalizuje, čili spočítá se jeho velikost, kterou se vydělí všechny jeho složky.



Obr. 3.10.: určení koordinátů bodu P

Nyní je možné z normalizovaného vektoru dopočíst oba dva potřebné body *U* a *V*. Jelikož je potřeba dodržet požadavek, týkající se omezení možných výsledných hodnot, pak jsou oba body vyděleny nejvyšší možnou hodnotou, kterou mohou nabývat odpovídající úhly. Tím se docílí, že výsledky jsou v rozpětí -0.5 až 0.5, a proto je vždy přičtena hodnota 0,5, čímž je docíleno splnění podmínky:

$$U = \frac{\cot\left(\frac{x}{z}\right)}{2\pi} + \frac{1}{2}$$

$$V = \frac{\sin^{-1} y}{\pi} + \frac{1}{2}$$

3.5. Odraz od vodní hladiny

Pokud je ve scéně zobrazena voda, pak je možné využít odrazu terénu od vodní hladiny, které může vést k celkově mnohem lepšímu vizuálnímu vzhledu, ovšem za cenu snížení rychlosti vykreslování. Příčinou zpomalení je nutnost objekt, který se odráží, vykreslovat dvakrát; jednou jako originál, podruhé pak jeho odraz.

3.5.1. Stencil Buffer

Slouží k určování, které z fragmentů se budou vykreslovat a které nikoliv. Ve výsledku tedy určuje, že na zamaskovaných místech se neprovádí vykreslování.

Uplatnění stencil Bufferu je právě především při použití některých grafických efektů, jakými mohou být zrcadlení či stíny.

V každém cyklu vykreslování je nutné nejprve vymazat obsah stencil bufferu spolu s obsahem depth bufferu a color bufferu. Součástí jeho aplikování je také zavedení testovací funkce, která porovnává referenční hodnotu s maskou. Pro každý pixel se tak provede operace AND referenční hodnoty a pixelem uloženým v stencil bufferu a na základě výsledku testu je změněna hodnota v bufferu.

Pro odrazy je obvykle požadováno, aby se objevili například na nějaké lesklé ploše, zrcadlu či vodní hladině. Je třeba tuto plochu definovat, tak že pro každý pixel, na kterém je zobrazena, vyhodnotí testovací funkce jako 1 v stencil bufferu.

Obrázek 3.11 ukazuje možnou výslednou situaci, kdy bílé místo s hodnotou 1 značí oblast, na které bude docházet k odrazu, zbytek byl vyhodnocen hodnotou 0 a je znázorněn černou barvou.

Obr. 3.11.: Znázornění obsahu stencil bufferu

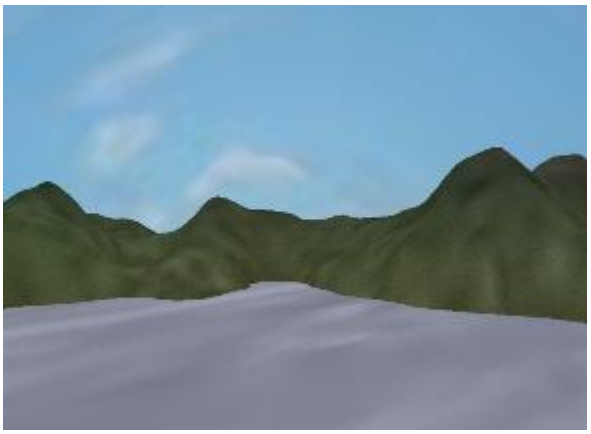
Pro správnou funkčnost je také nutné povolit stencil bufferu a v případě, kdy již není potřebný jej naopak zakázat pomoci funkcí glEnable [6] a glDisable [6].

3.5.2. Postup při vytváření odrazu

Je velmi důležité, aby se odražený terén vykresloval jen na pixelech, na kterých se nachází vodní plocha, neboť by hrozilo při náhodném manipulování s kamerou, že by byl vykreslen právě i na místech, které jsou pro umístění odrazu naprosto nereálná. Právě proto je použit stencil buffer, jehož princip spočívá v tom, že je nejprve naplněn samými nulami a následně v místech, kde se vodní plocha nachází vyplněna jedničkami. Pouze v oblasti, kde jsou jedničky, pak bude vykreslován odraz.

Postup je následující:

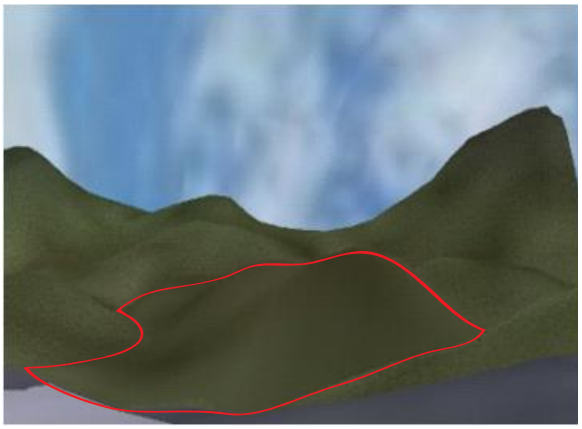
- Vykreslení vodní hladiny do stencil bufferu
- Vykreslení odrazu terénu, přičemž je nutné jej zrcadlově otočit (y-ové souřadnici je změněno znaménko)
- Vykreslit vodní hladinu
- Vykreslit terén



Obr. 3.12.: výsledný obraz bez odrazu

Obr. 3.13.: výsledný obraz s odrazem

V případě, kdy originální terén zasahuje i pod vodní hladinu, je zřejmé, že odražený terén naopak zasahuje nad povrch vody, jak objasňuje obrázek 3.14. Proto je nutné tento jev odstranit.



Obr. 3.14.: odražený terén vystupující nad vodní hladinu

Dalším důležitým krokem tedy je ořezávat ty části odraženého terénu, které přečnívají nad vodní hladinu. Opět je možné využít funkci z OpenGL, jež umožní ořezávání pomoci určité rovnice, kterou je nutné nejprve, dle požadované funkčnosti, nadefinovat.

3.6. Modely

Samotný povrch terénu je však nedostačující k vytvoření iluze reálné krajiny, a proto je vhodné rozšířit aplikaci o možnost přidávání různých modelů představující například přírodní objekty, jakými mohou být například stromy. Lze ale používat také modely domů či jakékoliv jiné objekty reálného světa.

3.6.1. OBJ formát

OBJ je ASCII formát 3D modelu, se kterým lze díky jeho jednoduchosti velmi dobře pracovat. Zároveň jej lze exportovat pomoci většiny oblíbených grafických programů 3D studio Max, Blender a mnoha dalších.

Soubor je koncipován tak, že první znak na každém řádku určuje příkazy, z nichž ty nejdůležitější jsou v tabulce 3.1 a další znaky náležící stejnému řádku jsou argumenty. Čtení a parserování probíhá po řádcích od prvního řádku k poslednímu.

OBJ formát a další ASCII formáty všeobecně obvykle slouží k možnosti vytvoření vlastního formátu modelu, neboť lze do takovéhoto souboru jednoduše nahlédnout, zjistit jeho strukturu a vybrat si jen data, která jsou v dané chvíli důležitá.

Příkaz	Funkce
# komentář	Komentář bývá často při načítání ignorován. Obvykle například první řádek souboru říká, jakým programem byl soubor OBJ vygenerován.
v *x y z*	Řádek vertex, resp. bodu v prostoru, jehož tři argumenty udávají jeho souřadnice.
vt *u v [w]*	Definuje texturové koordináty UV (parametr W je volitelný). Argumenty nabývají hodnot v rozmezí 0 až 1.
vn *x y z*	Normálový vektor (kolmý vektor). Argumenty udávají souřadnice.
f *v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3*	Specifikuje polygon, jehož jednotlivé argumenty definují indexy bodů, texturových koordinátů a normál, ze kterých se skládá.

Tab. 3.1.: Tabulka příkazů a jejich významu používaných formátem OBJ

3.6.2. Osvětlení modelu

OpenGL používá pro výpočet barev těles Phongův osvětlovací model [1]. Na rozdíl od fyzikálně přesnějších modelů je navržen tak, aby byl výpočet co možná nejrychlejší a zároveň aby poskytoval přirozené výsledky.

VPhongově osvětlovacím modelu se světlo, které na určitý objekt dopadá a následně odráží, se rozkládá na tři světelné složky:

- **Ambientní** – okolní světlo, osvětlující model ze všech směrů bez závislosti na poloze zdroje.

$$I_a = I_A \cdot r_a \,,$$

  kde I_A vyjadřuje intenzitu okolního světla stejnou pro celou scénu. Koeficient r_a je koeficientem odrazu okolního světla vyjadřující schopnost povrchu odrážet ambientní světlo.
- **Difuzní** – světlo, které na objekt dopadá ze zdroje a od povrchu se odráží do všech směrů o stejné intenzitě.

$$I_d = I_L \cdot r_d(\vec{l} \cdot \vec{n}) \,,$$

  kde I_L značí barevné složení dopadajícího paprsku, r_d je koeficient difúzního odrazu. Vektor l je jednotkovým vektorem pohledu a vektor n je pak symetrický vektor s l podle normály.
- **Zrcadlový** - odlesky vznikají ve chvíli, kdy na povrch dopadá světelný paprsek odrážející se podle zákonu odrazu a lomu.

$$I_s = I_L \cdot r_s(\vec{v} \cdot \vec{r})^h \,,$$

  Kde I_L opět značí barevné složení dopadajícího paprsku, r_s je koeficient zrcadlového odrazu, vektor v je jednotkovým vektorem pohledu, vektor r je symetrický vektor s v podle normály a skalární koeficient h vyjadřuje ostrost zrcadlového odrazu.

Dopadá-li na určitý bod paprsek z N světelných zdrojů, pak platí:

$$I_V = I_A \cdot r_a + \sum_{i=1}^{N} I_{L_k}[r_s(\vec{v}\cdot\vec{r_i})^h + r_d(\vec{l_i}\cdot\vec{n})]$$

OpenGL verze 2.1 umožňuje nastavit až osm světelných zdrojů, ale je důležité vědět, že více zdrojů světla má vliv na výkon celé aplikace, protože dochází samozřejmě k složitějším výpočtům osvětlení a tím i k poklesu rychlosti vykreslování.

Osvětlení modelu a s ním související stínování je velmi důležitou součástí scény, jinak by bylo velmi obtížné vnímat tvar objektu. Bez použití osvětlení by se na celý povrch tělesa nanesla jen jedna jediná barva, čímž by bylo možné rozeznávat pouze obrys tělesa.

OpenGL používá konstantní nebo Gouraudovo stínování. Konstantní stínování je nejjednodušší a patrně i nejrychlejší metoda. Předpokládá, že každá ploška má právě jednu normálu, podle které je vypočítána barva, jež se nanese na celou plochu. Nevýhodou je, že u oblých povrchů zdůrazňuje, že těleso je ve skutečnosti aproximováno určitým množstvím plošek.

Naopak Gouraudova metoda zajišťuje plynulý barevný přechod u zakřivených povrchů tak, že jednotlivé plochy, ze kterých se objekt skládá, jsou nezřetelné. Principem je, že jsou známy barevné odstíny všech vrcholů plochy. Poté jsou vypočítány barevné odstíny vnitřních bodů pomoci bilineární interpolace. Pracuje však pouze s ambientní a difúzní složkou, nikoliv se zrcadlovou vzhledem k metodě výpočtu.

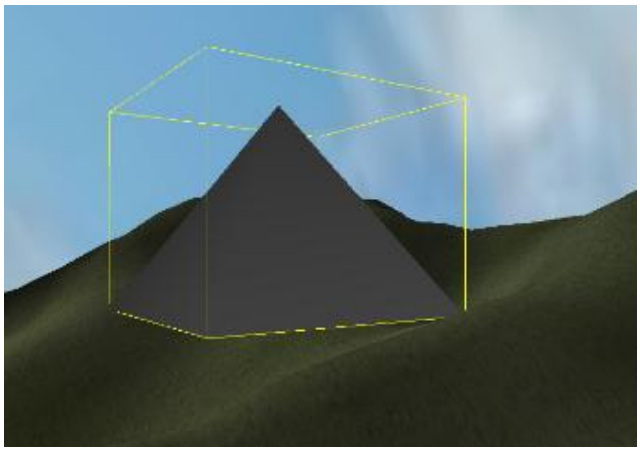
Pokud je kladen požadavek na osvětlení modelu, je nezbytné vypočítat či ze souboru načítat normály, která se musí bezpodmínečně zadávat u každého vrcholu, aby se mohla korektně vypočítat

barva tělesa. Dále je nutné normálu pro korektní funkčnost výpočtu normalizovat, čili délka vektoru musí být jednotková.

3.6.3. Bound box

Obvyklým požadavkem na modely je umožnit s nimi jistou manipulaci. Ať už jde o mazání, přesouvání či jiné interakce, je nejprve nutné objekt nějakým způsobem identifikovat, resp. rozpoznat s jakým modelem je pracováno. Jedna z možností je procházet a kontrolovat pozice všech polygonů, ze kterých se model skládá. Pokud by však byl model velmi komplikovaný a skládal se z velkého množství polygonů, mohlo by dojít k výraznému poklesu rychlosti aplikace.

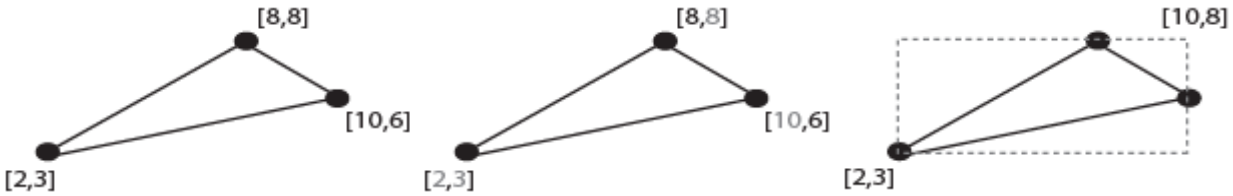
Právě proto se celý objekt obaluje kvádrem (bound box) jako je možno vidět obrázku 3.15 a veškeré testování se omezuje na testování s touto obálkou objektu, což vede ke značnému zrychlení, protože při testování nezáleží na počtu polygonů, ze kterých se model skládá.



Obr. 3.15.: boundbox kolem modelu

Tvorba bound boxu může probíhat již při načítání jednotlivých vertexů, kdy jsou vybrány jednotlivé složky ze všech souřadnic s nejmenšími a největšími hodnotami. Takto jsou určeny dvě souřadnice, pomoci kterých je možno již snadno vytvořit krychli.

Princip vytváření bound boxu je stejný i v případě rovinných objektů, kdy jediným rozdílem je porovnání pouze dvou souřadnic. Konkrétní příklad vypočítání obalu objektu je na obrázku 3.16, kdy se nejprve postupně projdou všechny souřadnice bodů a proběhne porovnání jejich složek a tím získání minimální a maximální souřadnice trojúhelníku.



Obr. 3.16.: Vytváření bound boxu

Takto se získají dvě souřadnice, které lze použít pro vytvoření obalu, který ohraničuje celý objekt. Průběh porovnání je znázorněn v tabulce 3.2:

Porovnávaná souřadnice	Nejmenší souřadnice	Největší souřadnice
[2,3]	[2,3]	[2,3]
[8.8]	[2,3]	[8,8]
[10,6]	**[2,3]**	**[10,8]**

Tab. 3.2.: Vybrání minimální a maximální souřadnice

3.7. Průchod terénem

Důležitou součástí každé grafické aplikace je také určitá možnost procházet vytvořený virtuální svět. K tomu slouží kamera, což ve skutečnosti není nic jiného než aplikování transformací posunu či rotace v prostoru s možností poskytnout ovládání uživateli.

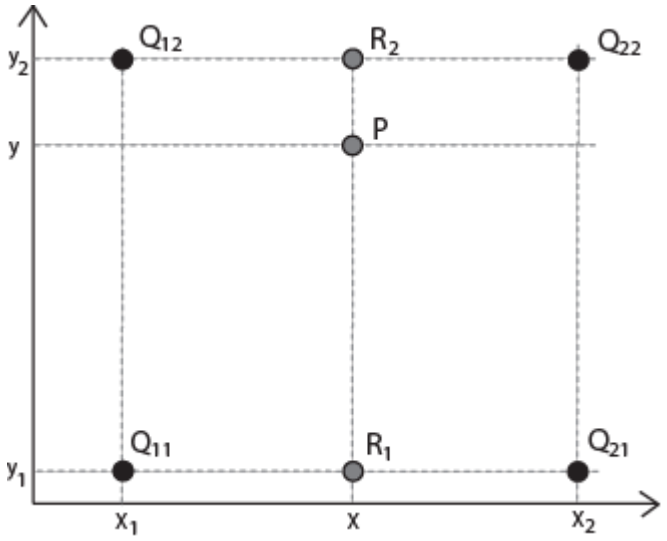
Lze využít funkci *gluLookAt* [6], jež poskytuje OpenGL, resp. její rozšíření GLU. Tato funkce přebírá argumenty aktuálního vektoru pozice kamery, vektor směru pohledu a směr UP vektoru.

Při každém vykreslovacím cyklu je mimo jiné volána funkce *gluLookAt* a je tak vždy aktualizována pozice a výhled pozorovatele.

V případě, že bude umožňováno chodit po terénu, je nutné zajistit, aby kamera kopírovala povrch. Využije se souřadnic jednotlivých bodů v závislosti na aktuální pozici kamery. Jelikož vzdálenost mezi body v rovině XZ je poměrně velká, bude nutné využít bilineární interpolaci, jinak by mohlo dojít k průchodu skrz povrch.

Bilineární interpolace vychází z interpolace lineární, kdy je provedena nejprve interpolace na x-ové a následně na y-ové(popřípadě z-ové) souřadnici. Jelikož je terén tvořen trojúhelníkovými pasy, lze snadno získat souřadnice všech čtyř bodů nutných pro výpočet právě dle aktuální pozice, na které se pozorovatel nachází.

Jsou tedy známy souřadnice bodů a zároveň jejich funkční hodnota $Q_{11}=(x_1, y_1)$, $Q_{12}=(x_1, y_2)$, $Q_{21}=(x_2,y_1)$, $Q_{22}=(x_2, y_2)$, z nichž je nyní možné určit přibližné souřadnice bodu P. Situaci ilustruje obrázek 3.17.



Obr. 3.17.: bilineární interpolace

Interpolace na x-ové ose:

$$f(R_1) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) \approx \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

Interpolace na y-ové ose:

$$f(P) \approx \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2)$$

4. Optimalizace

Vykreslování 3D grafiky je výpočetně velmi náročné a klade velké nároky na hardwarové vybavení počítače. Přitom je velmi důležité zajistit, aby byla zajištěna plynulost grafiky.

Obecně braná hodnota, při které lidské oko nedokáže rozeznat jednotlivé obrazy, je zhruba 25 FPS (frames per sekond). V případě klesání pod tuto hranici, začne docházet k „sekání“ obrazu a člověk přestává mít dojem spojité animace.

Aby se trhání zabránilo, používají se obvykle různé optimalizační algoritmy, jejichž úkolem je omezit počet trojúhelníků, které se vykreslují. Dochází k ořezávání těch trojúhelníků, které pozorovatel v daný okamžik nevidí, neboť se nenacházejí v pozorovatelově zorném poli.

Přesněji vlastní ořezávání viditelných a neviditelných objektů si řeší grafická karta sama. Optimalizacemi, jež jsou zde naznačeny, dochází ke zmenšení množství dat, které se při vykreslování musí posílat přes sběrnici počítače do grafické karty.

Není však možné testovat všechny trojúhelníky a zjišťovat, které mají být vykresleny a které nikoliv. Nespornou výhodou by samozřejmě bylo to, že se vykreslí skutečně jen trojúhelníky, které jsou viditelné. Na druhou stranu zkoumání viditelnosti každého z velkého množství trojúhelníků by bylo časově náročnější než vykreslování všech trojúhelníků.

4.1. Back-face culling

Back-face culling je pravděpodobně nejjednodušší technikou, jakou lze v některých případech zrychlit grafickou aplikaci odstraněním odvrácených ploch, čímž se zmenší počet vykreslovaných trojúhelníků. Pokud aplikace navíc využívá rozhraní OpenGL, pak je snadná i implementace, kdy stačí pouze Back-face culling povolit.

Samozřejmě lze algoritmus použít i v aplikacích, které nejsou podporovány OpenGL a tomto případě je nutné jej celý implementovat.

Pro každý trojúhelník, který se nachází ve scéně, se spočítá normála, tj. vektor kolmý na plochu trojúhelníku. Následně se spočítá skalární součin vektoru kamery a normály. Pokud je výsledek součinu menší než nula, znamená to, že trojúhelník je přivrácený k pozorovateli a je možné jej vykreslit.

Pseudokód algoritmu Back-face culling:

```
for each trojuhelnik
{
    Vektor v1 = trojuhelnik.bod3 – trojuhelnik.bod1;
    Vektor v2 = trojuhelnik.bod3 – trojúhelník.bod2;
    Vektor normal = VypoctiKolmyVektor(v1, v2);
    float uhel = SkalarniSoucin(vektor_kamery, normal);
    if uhel < 0 then
    {
        Trojuhelnik.vykresli();
    }
    else
    {
        continue;
    }
}
```

4.2. Quad-Tree

Algoritmus Quad-Tree přímo optimalizaci a viditelnost sám o sobě neřeší, nýbrž jen seskupuje sousední polygony do určitých oblastí, které jsou ohraničeny krychlemi. Při testování viditelnosti se pak zjišťuje, které krychle jsou ve výhledu kamery a příslušné polygony, nacházející se ve viditelných krychlích, budou vykresleny.

Vytváření stromu probíhá tak, že je nejprve vytvořen kořen, resp. krychle, která obaluje celý objekt. Proto veškeré polygony, na které je algoritmus aplikován, patří do kořene.

Jak už název napovídá, jedná se o datovou strukturu stromu, kdy každý uzel má čtyři potomky. Tudíž v dalším kroku se vytvoří čtyři další krychle, jež jsou potomky kořene. Takto se bude každá krychle dále rekurzivně dělit vždy na další čtyři potomky. Dělení bude pokračovat tak dlouho, dokud bude splněná podmínka umožňující další dělení aktuálně testovaného uzlu.

Pseudokód rekurzivní funkce vytvářející stromu:

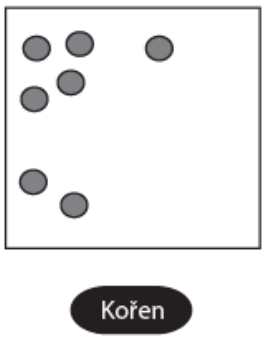
```
void vytvoř_uzel(Uzel rodič)
{
    if splněná podmínka
    {
        for each potomek
        {
            urči střed nového uzlu;
            urči velikost nového uzlu;
            vytvoř_uzel(rodič.potomek);
        }
    }
    else
    {
        return;
    }
}
```

Podmínkou pro ukončení dělení bývá obvykle limit počtu polygonů, který se musí nacházet v testovaném uzlu, aby byl rozdělen. Je-li množství polygonů větší než zadaný limit, pak je uzel dále rozdělen. V opačném případě se tento uzel stává listem, tj. uzlem, který nemá potomky.

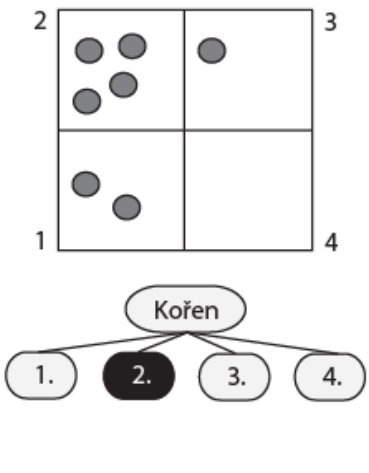
Často se také používá podmínka ve smyslu maximální hloubky stromu. Pokud se dosáhne stanovené hloubky stromu, další dělení pak také neproběhne.

Vytváření stromu je znázorněno na obrázcích 4.1 - 4.3, kde je podmínkou pro rozdělení maximální možný počet koleček v uzlu, který je stanoven na 3, tzn., že pokud se v uzlu nachází více než 3 kolečka, bude rozdělen.

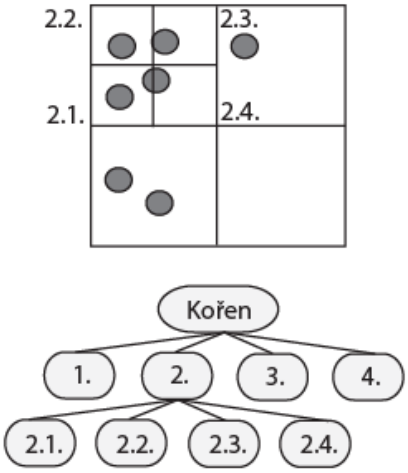
Na obrázku 4.1 je vytvořen kořenový uzel. Předem definovaný prostor, na kterém je algoritmus aplikován se celý nachází v kořeni stromu. Jelikož se zde nachází 7 koleček, dojde k rozdělení kořene na čtyři potomky. Nastalá situace je ilustrována na obrázku 4.2 a následně jsou všechny nově vytvořené uzly znovu testovány na podmínku omezující maximální počet objektů. Obrázek 4.3 ukazuje další dělení uzlu a konečný strom.



Obr. 4.1.: vytvoření kořene



Obr. 4.2.: rozdělení uzlu na potomky



Obr. 4.3.: finální strom

Z uvedených vlastností je zřejmé, že vytvořený strom nemusí míst listy na stejné úrovni. Jelikož je terén, na který se quad-tree aplikuje, tvořen pravidelnou sítí, pak tomu tak ovšem je. Pokud by se ale algoritmus aplikoval na nepravidelný objekt, pak by v místech s větším shlukem polygonů, bylo pravděpodobně vybudováno více uzlů než v místech s jejich menším počtem. Evidentně však záleží na podmínce, která dělení ukončuje.

Mohou nastat i situace, kdy použití quad-tree algoritmu aplikaci neurychlí. Dochází k tomu v případě, kdy je viděn celý objekt, na který byl algoritmus použit. To znamená, že dojde dokonce ke zpomalení vykreslování. Nezahrne-li se do celkového zpomalení samotné vytvoření stromu, pak ke značnému zpomalení může vést zbytečné rekurzivní procházení stromu a zjišťování, které uzly jsou viditelné.

Druhou velkou slabinou tohoto algoritmu je špatně zvolená podmínka ukončení dělení uzlů. Nemá význam, aby byl počet trojúhelníků v uzlu zbytečně malý, čímž by došlo k velkému nárůstu uzlů, což vede k prodloužení průchodu stromu a také k většímu počtu testování, které uzly jsou viditelné.

Nevýhodu, jež plyne ze špatné ukončovací podmínky, lze však velice jednoduše odstranit. Není problém předem zjistit, z kolika polygonů se scéna skládá a na základě této hodnoty určit rozumný limit, při kterém dojde k rozdělení uzlu.

Existuje i varianta datové struktury pro dělení objektu, kdy každý uzel má osm potomků. Jedná se o OcTree(octal tree). Pro terén však použití této struktury není příliš vhodné z důvodu menšího výškového rozdílu. Není tudíž nutné rozdělovat terén ještě horizontálně.

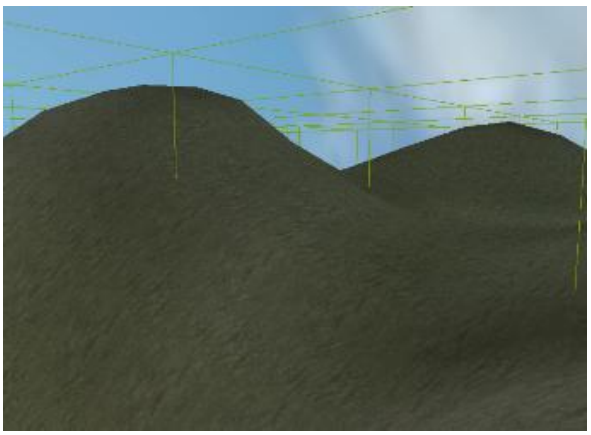
Mimoto se používá celá řada dalších struktur, některé jsou ovšem pro urychlení zobrazování terénu nevhodné. Naopak jejich možnosti optimalizace je možno využít například u grafických aplikací, které znázorňují interiérové scény.

4.3. Frustum culling

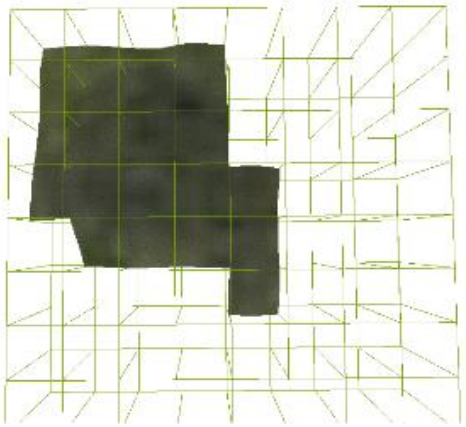
Frustum culling je technika, pomoci které lze výhledem kamery ořezávat scénu. Výhled kamery si lze velmi dobře představit jako čtyřboký komolý jehlan, který je tvořen šesti ořezávacími rovinami.

Algoritmus quad-tree rozdělil terén do oblastí, které jsou vymezeny krychlemi. Nyní pomoci frustum culling je potřeba otestovat každou vytvořenou krychli se všemi šesti rovinami, zda je protínána či leží celá uvnitř výhledu. Pokud test skončil úspěšně, pak jsou vykresleny všechny

trojúhelníky, které se v dané krychli nacházejí. Výsledek ořezání jednotlivých pro kameru neviditelných uzlů je viditelné na obrázku 4.5, který jasně ukazuje, které uzly jsou vykreslovány v závislosti na pozici pozorovatele na obrázku 4.4. Viz příloha C.



Obr. 4.4.: pohled pozorovatele



Obr. 4.5.: ořezání pohledu

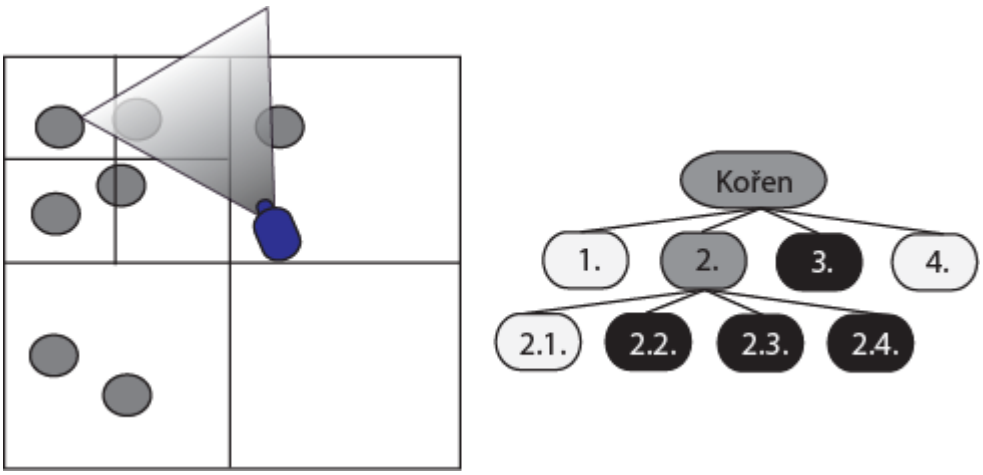
Vykreslování probíhá tak, že se nejprve začíná u kořenového uzlu. Vždy pokud je některý z aktuálně testovaných uzlů viditelný, pak stejným způsobem budou přezkoumány všichni jeho potomci. V opačném případě se celá větev vynechá, neboť není-li viditelný rodič, pak je zřejmé, že nemohou být viditelní ani jeho potomci.

Testy jsou prováděny rekurzivně, a pokud se dojde až k uzlu bez potomků, kdy se zjistí, že je viditelný, pak je tento listový uzel vykreslen. U vnitřních uzlů se vykreslování neprovádí, vždy je při pozitivních testech nutné projít strom až k listům.

Pseudokód funkce pro vykreslování:

```
void VykresliUzel (Uzel uzel)
{
    if uzel je viditelný
    {
        if uzel má potomky
        {
            VykresliUzel(uzel.potomek[0]);
            VykresliUzel(uzel.potomek[1]);
            VykresliUzel(uzel.potomek[2]);
            VykresliUzel(uzel.potomek[3]);
        }
        else
        {
            Vykresli obsah uzlu;
        }
    }
    else
    {
        return;
    }
}
```

Pro strom z obrázku 4.3 lze průchod stromu v závislosti na pozici kamery znázornit jako na obrázku 4.6, kde šedě vybarvené uzly jsou uzly viditelné a černě vybarvené uzly jsou uzly viditelné a jejichž obsah je vykreslen, kdy je jasně patrné, že se vykreslují právě listové uzly.



Obr. 4.6.: průchod stromu

5. Implementace

Implementace byla provedena v jazyce Java především proto, aby bylo možné program spustit jako Java applet v internetovém prohlížeči. K vykreslování trojrozměrné grafiky pak bylo využito knihovny JOGL, především kvůli podobnosti s programováním pomoci OpenGL v jazyku C nebo C++.

Výsledná aplikace nabízí možnost vygenerovat terén pomoci popsaných algoritmů. Uživateli je umožněno vkládat pomoci jednoduchého grafického uživatelského rozhraní vstupní hodnoty pro jednotlivé algoritmy a také může volit velikost povrchu krajiny. Dále je dovoleno manipulovat s pozicí světelného zdroje či ovlivňovat podmínku ukončení dělení quadtree ve smyslu minimálního počtu trojúhelníků v uzlu. Při generování terénu se provedou následující činnosti:

- Kontrola vstupů zadaných uživatelem
- Vygenerování výškové mapy vybraným algoritmem
- Vyhlazení povrchu
- Nastavení vodní hladiny
- Vytvoření textury povrchu terénu
- Vytvoření skydomu v závislosti na rozměrech krajiny
- Vytvoření quadtree
- Vygenerování textur

Pro prezentaci jednotlivých implementovaných funkcí, týkajících se zejména vykreslování, je možné jejich zapínání a vypínání. Je tak velmi snadno docíleno nejen regulace mezi výkonem a kvalitou výsledného obrazu, ale především názorné ukázky použitých algoritmů a technik. S tímto úzce souvisí i používání dvou kamer, kdy jedna má funkčnost jako hlavní kamera pozorovatele, na kterou je aplikován frustum culling a dochází tedy k ořezávání neviditelných oblastí. Zatímco druhá kamera je zde použita za účelem jednoduchého náhledu uživatele, jak vypadá scéna po ořezání. Funkčnost ořezávání je tedy ověřitelná nejen zvýšením FPS, ale také tímto systémem dvou kamer.

K uložení výškových bodů krajiny je použita výšková mapa, která je celá uložená v paměti. Poskytuje však snadný přístup k jednotlivým bodům, jak při generování, filtrování, tak i při operacích, které se týkají vykreslování.

Vygenerovaný terén je možné také vyhladit a následně je výsledek zobrazen. Povrch krajiny je vykreslován pomoci trojúhelníkové sítě, na kterou je posléze mapována textura povrchu, popřípadě i textura znázorňující zdrsněný povrch a lightmapa.

Vykreslovací cyklus:

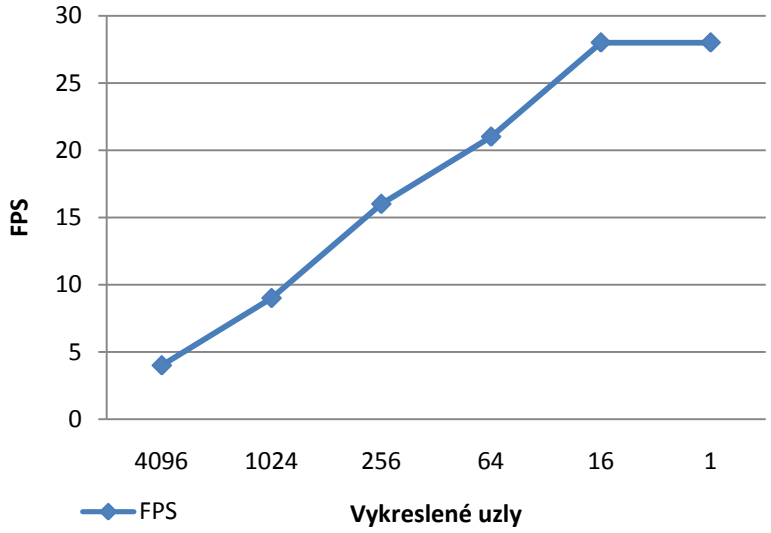
- Aktualizace kamery
- Výpočet frustum cullingu
- Vykreslení viditelných uzlů
    - Vykreslení ohraničení uzlů
    - Vykreslení obsahu viditelných uzlů včetně texturování
    - Vykreslení vody a viditelných odražených uzlů
- Vykreslení umístěných modelů
- Vykreslení skydomu

Nicméně velkým problémem se při vývoji ukázalo používání textur, kdy bylo možné vytvářet textury voláním funkce pro jejich generování jen z hlavní třídy, která implementuje GLEventListener. Toto má za následek složitější přidávání textur za běhu aplikace, především v okamžiku, kdy je třeba nahradit stávající lightmapu novou z důvodu změny pozice světla.

Během implementace však bylo nutné myslet především na rozumné požadavky na hardware a to i přesto, že je použita grafická knihovna zpřístupňující hardwarově akcelerovanou grafiku. Je důležité rovnoměrně rozložit zátěž na operační paměť a procesor. Ne všechno je nutné v každém

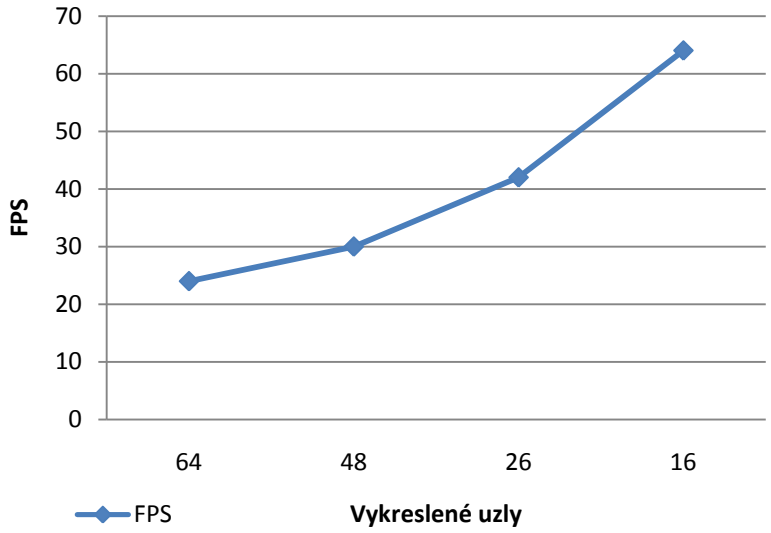
vykreslovacím cyklu přepočítávat, proto je vhodné mít některá často používána data uloženy v paměti. Stejně tak však nelze přeplnit operační paměť počítače.

Použití algoritmů, jež optimalizují scénu, má za výsledek díky snížení počtu vykreslených polygonů zvýšení rychlosti vykreslování. Graf 5.1 ukazuje hodnotu FPS při běhu aplikace s různě zvolenou velikostí uzlu s viditelným celým terénem v závislosti na počtech uzlů, které vykresluje. Je zde jasně patrné, jak quadtree může vést ke zpomalení vykreslování. Testování proběhlo na terénu vygenerovaným metodou přesouvání středového bodu s rozlišením 256x256. Celkově je vykreslováno 131072 trojúhelníků.



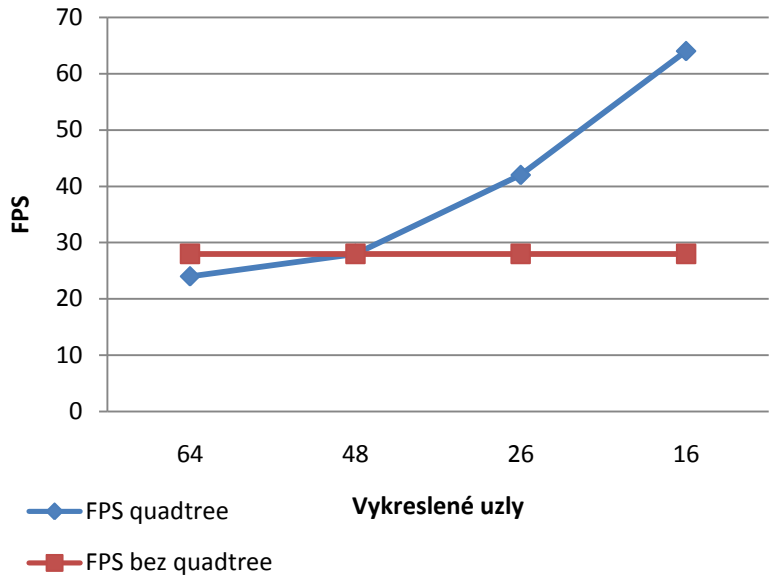
Graf 5.1.: závislost FPS na počtu vykreslených uzlech u kompletně viditelného terénu

Uživatel obvykle nemá možnost při procházení krajiny vidět celý terén, čili vidí jen část celkového modelu. Graf 5.2 tak ukazuje závislost FPS na vykreslených uzlech, kdy pozorovatel prochází terén.



Graf 5.2.: závislost FPS na počtu vykreslených uzlech u procházení terénu

A konečně graf 5.3 ukazuje srovnání FPS v aplikaci při vykreslování pomocí quadtree a bez jeho použití, kdy je rychlost konstantní.



Graf 5.3.: srovnání vykreslování pomoci quadtree a bez jeho použití

6. Závěr

V této práci jsou popsány metody, jak lze procedurálně vygenerovat terén a také možnosti jeho zobrazení pomoci grafické knihovny OpenGL. Také jsou zde popsány implementované algoritmy, jež optimalizují scénu.

Výsledkem praktické části se tak stal applet, který používá algoritmy, jež jsou v této práci teoreticky popsány. Aplikace dokáže pomoci uživatelem zvoleného algoritmu vygenerovat výškovou mapu, vytvořit textury povrchu a následně terén zobrazit. Lze ji tedy použít jako ukázkovou aplikaci pro demonstraci jednotlivých algoritmů.

Aplikace má velké možnosti rozšíření a to jak při generování krajin implementací erozních algoritmů, tak také v oblasti vykreslování a optimalizací. Techniky LOD (level of detail) terénu by výrazně urychlily vykreslování ve chvíli, kdy by byla viděna velká část či dokonce celá krajina. Stejný postup by bylo možné využít i u modelů, kdyby pro jeden objekt bylo vytvořeno více modelů s různým stupněm detailu a podle vzdálenosti se použil odpovídající model. Také je velmi rozšířeno aplikování pixel a vertex shaderů do grafických aplikací, které by bylo možné také použít.

Literatura

[1] ŽÁRA, Jiří; BENEŠ, Bedřich; SOCHOR, Jiří; FELKEL Petr. *Moderní počítačová grafika*. Vyd. 1. Brno : Computer Press, 2004. 609 s. ISBN 80-251-0454-0.

[2] SHREINER, Dave; WOO, Mason; NEIDER, Jackie; DAVIS, Tom. *The OpenGL Programming Guide* [online]. Addison Wesley Publishing company [cit. 2010-03-30]. Dostupné z WWW: <http://www.glprogramming.com/red/>.

[3] MARTZ, Paul. *Gameprogrammer* [online] [cit. 2010-03-30]. Generating Random Fractal Terrain. Dostupné z WWW: <http://www.gameprogrammer.com/fractal.html>.

[4] TIŠNOVSKÝ, Pavel. Trojrozměrné modely terénu. *Root.cz* [online]. 3.10.2006,[cit. 2010-04-07]. Dostupný z WWW: <http://www.root.cz/clanky/trojrozmerne-modely-terenu/>.

[5] FERNANDES, António Ramires. *Lighthouse3d* [online][cit. 2010-03-30]. Artificial Terrain Generation. Dostupné z WWW: <http://www.lighthouse3d.com/opengl/terrain/index.php3?fault>.

[6] *Opengl.org* [online]. [cit. 2010-04-07]. OpenGL 2.1 Reference Pages. Dostupné z WWW: <http://www.opengl.org/sdk/docs/man/>.

[7] *Java™ Binding for the OpenGL® API Wiki* [online]. [cit. 2010-04-07]. Java™ Binding for the OpenGL® API Wiki. Dostupné z WWW: <http://kenai.com/projects/jogl/pages/Home>.

[8] *Microsoft* [online]. [cit. 2010-04-07]. Microsoft DirectX. Dostupné z WWW: <http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx>.

[9] *OpenGL.org* [online]. 2000-11-11 [cit. 2010-04-07]. OpenGL.org. Dostupné z WWW: <http://www.opengl.org/>.

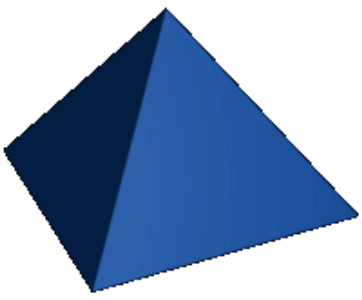
[10] TIŠNOVSKÝ, Pavel. Fraktály v počítačové grafice. *Root.cz* [online]. 2006, [cit. 2010-04-07]. Dostupný z WWW: <http://www.root.cz/serialy/fraktaly-v-pocitacove-grafice/>.

Příloha A

Formát OBJ

Soubor pyramida.obj:

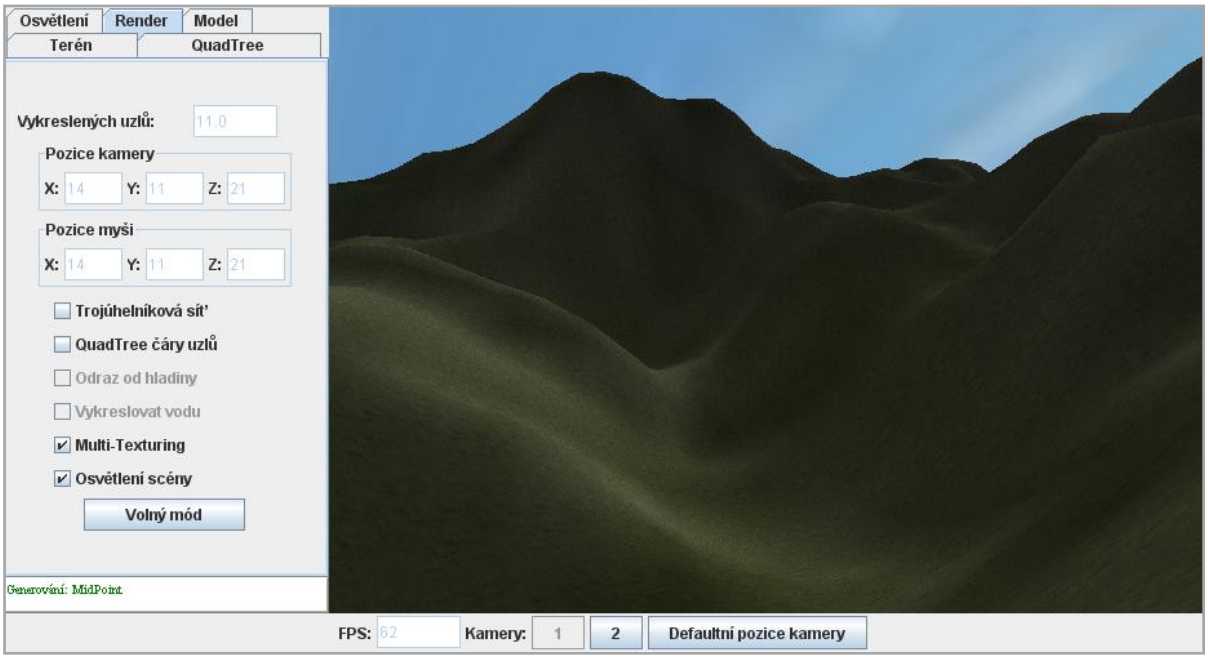
```
# Max2Obj Version 4.0 Mar 10th, 2001
#
# object Pyramid01 to come ...
#
v -7.569088 28.296122 18.446186
v -29.928978 0.005000 39.570412
v 14.790802 0.005000 39.570412
v 14.790802 0.005000 -2.678040
v -29.928978 0.005000 -2.678040
v -7.569088 0.005000 18.446186
# 6 vertices

vt 0.500000 1.000000 0.000000
vt 0.000000 0.000000 0.000000
vt 1.000000 0.000000 0.000000
vt -0.029248 0.000000 0.000000
vt 0.970752 0.000000 0.000000
vt 0.000000 1.000000 0.000000
vt 1.000000 1.000000 0.000000
vt 1.000000 0.000000 0.000000
vt 0.000000 0.000000 0.000000
vt 0.500000 0.500000 0.000000
vt 0.500000 1.000000 0.000000
# 11 texture vertices

vn 0.000000 0.598293 0.801278
vn 0.000000 0.598293 0.801278
vn 0.000000 0.598293 0.801278
vn 0.784548 0.620068 -0.000000
vn 0.000000 0.598293 -0.801278
vn 0.000000 -1.000000 -0.000000
# 6 vertex normals

g Pyramid01
f 1/1/1 2/2/2 3/3/3
f 1/11/1 3/4/3 4/5/4
f 1/1/1 4/2/4 5/3/5
f 1/11/1 5/4/5 2/5/2
f 2/6/2 6/10/6 3/7/3
f 3/7/3 6/10/6 4/8/4
f 4/8/4 6/10/6 5/9/5
f 5/9/5 6/10/6 2/6/2
# 8 faces
```



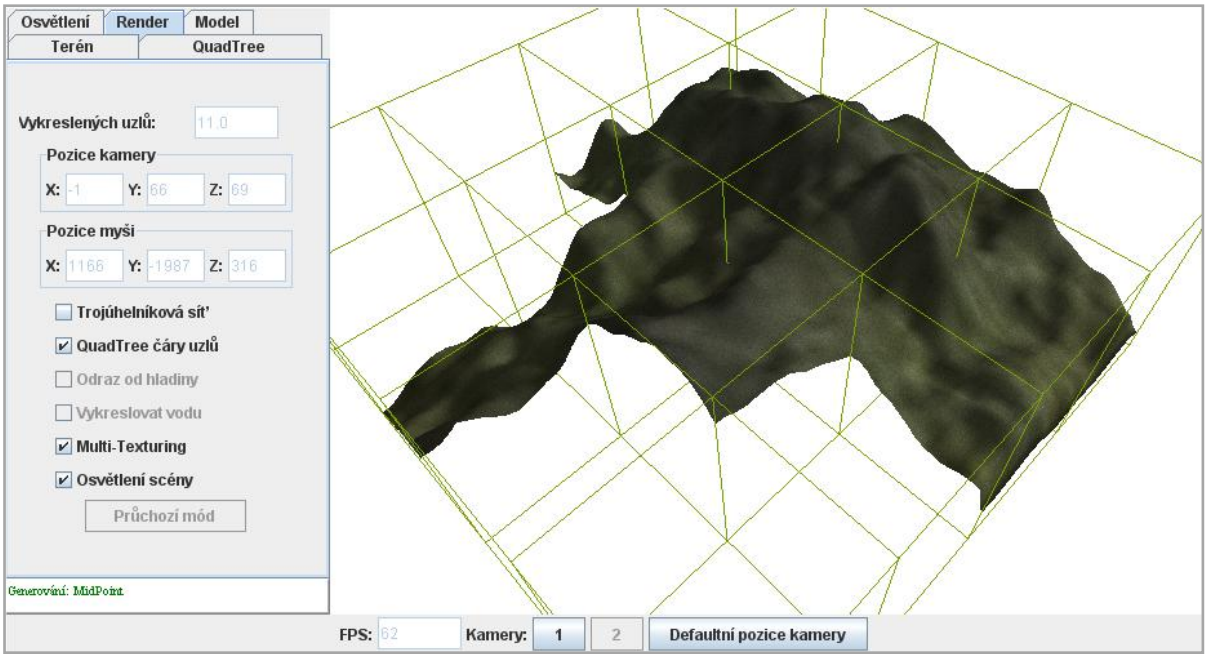
Obr. A.1.: Výsledný model

Příloha B

Náhled pomocí dvou kamer



Obr. B.1.: Pohled z kamery pozorovatele



Obr. B.2.: Pohled z druhé kamery znázorňující ořezání

Příloha C

Obsah přiloženého DVD

Na DVD jsou umístěny adresáře:

- **BcProject** - Projekt se zdrojovými kódy pro vývojové prostředí NetBeans
- **Applet** – Obsahuje HTML stránku se spustitelným appletem
- **Text** - Text bakalářské práce ve formátu docx, doc, pdf